Especialización en Modelado Conceptual: Un uso disciplinado de la Herencia*

Patricio Letelier† Pedro Sánchez† José A. Troyano‡ Yania Crespo§

† Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, {letelier, ppalma}@dsic.upv.es

‡ Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla, troyano@lsi.us.es

§ Departamento de Informática
Universidad de Valladolid, yania@infor.uva.es

Palabras Clave: Modelado Conceptual, Modelado Orientado a Objeto, Especialización, Herencia.

Resumen

En modelado conceptual los conceptos generalización y especialización tienen una estrecha relación con la noción de herencia ofrecida en lenguajes de programación orientados a objetos. Sin embargo, la herencia como mecanismo de programación tiene una aplicación más amplia. Además, existe una reconocida falta de consenso respecto de la interpretación y uso de la herencia. Esto, unido a la utilización de enfoques semiformales, hace difícil la transición desde un modelo conceptual con especialización hacia su correspondiente implementación. $\mathcal{OASIS}$ es un enfoque formal para el modelado conceptual orientado a objeto. En $\mathcal{OASIS}$ la especialización se provee a través del uso de constructores específicos del lenguaje. Estos constructores modelan directamente patrones de especialización en modelado conceptual basados en una semántica y sintaxis bien definidas. En este trabajo se exponen algunos de los aspectos más relevantes del tratamiento de la especialización en $\mathcal{OASIS}$.

1 Introducción

La especialización es un mecanismo de especificación que permite introducir información taxonómica en el modelo del sistema estableciendo un ordenamiento entre clases: unas más generales (superclases) y otras como especializaciones de las primeras (subclases) que heredan sus propiedades y posiblemente añaden nuevas más específicas. La mayoría de métodos para el modelado conceptual orientado a objeto incorporan la noción de herencia tal como se ofrece en lenguajes de programación. Esta situación queda ilustrada en la notación UML, donde la generalización (especialización) suele interpretarse como la herencia ofrecida por el lenguaje de programación utilizado para construir el sistema. La herencia, como mecanismo de programación, tiene una aplicación más amplia que la estrictamente asociada a la implementación de la especializa-

*Este trabajo está financiado por el proyecto *MENHIR* de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01.

ción, y además, es reconocida la falta de consenso respecto de su significado y uso [13]. La carencia de un mecanismo de especialización específico para el modelado conceptual dificulta la especificación sistema. Esta situación sumada a la utilización de métodos semiformales para el modelado conceptual hace más complicada la transición desde modelos conceptuales que incluyen especialización hacia sus correspondientes implementaciones.

La especialización y la herencia son mecanismos distintos, la especialización es útil para concebir un modelo conceptual y la herencia para implementarlo en un determinado lenguaje de programación orientado a objeto. Sin embargo, a pesar de ser distintos hay una estrecha dependencia entre ellos. Esta conexión aparece al intentar compaginar las características de la superclase y la clase especializada (la subclase) ya que necesitamos aplicar cierto tipo de herencia que nos permita combinar ambas especificaciones.

$\mathcal{OASIS}$ (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [7] es un enfoque formal para la especificación de modelos conceptuales siguiendo el paradigma orientado a objetos. En lo que respecta a la especialización, decimos que nuestro enfoque está caracterizado por un uso disciplinado de la herencia. La semántica de la herencia en $\mathcal{OASIS}$ está totalmente determinada por los constructores de especialización del lenguaje, los cuales fueron específicamente diseñados para abordar las situaciones de interés en modelado conceptual.

En este trabajo se presentan los aspectos más relevantes de la especialización como mecanismo de modelado conceptual dentro del marco de $\mathcal{OASIS}$. El objetivo es mostrar una perspectiva de la herencia en el ámbito del modelado conceptual, mostrando cómo dicha perspectiva está integrada en un lenguaje de modelado conceptual formal y sugiriendo que estas ideas pueden ser trasladadas a enfoques semiformales como UML.

Este documento se organiza en seis secciones, de las cuales esta introducción es la primera. La segunda sección se dedica a una breve presentación de $\mathcal{OASIS}$. En la tercera, que constituye el núcleo del trabajo, se presentan los mecanismos que proporciona $\mathcal{OASIS}$ para modelar especialización. En la cuarta sección se presenta un ejemplo aplicando los expuesto en la sección anterior. En la quinta sección se comentan algunos trabajos relacionados. Por último en la sexta sección se extraen conclusiones y las posibles vías de continuación.

2 Aspectos básicos de $\mathcal{OASIS}$

En $\mathcal{OASIS}$, un **objeto** es un proceso observable cuya vida está caracterizada por la ocurrencia de acciones, tanto si son solicitadas como si son recibidas por el objeto. Así, un objeto puede actuar como cliente o como servidor según esté solicitando u ofreciendo servicios.

Los servicios que un objeto proporciona a nivel "atómico" se denominan **eventos**. Cada objeto tiene un evento de creación (que inicia su vida) y de manera opcional uno de destrucción. Una **acción** es una tupla formada por el cliente, el servidor y el servicio solicitado. En la vida de un objeto específico, las acciones cuyo cliente es el mismo objeto son acciones solicitadas. Las acciones cuyo servidor es el mismo objeto son acciones servidas. Los eventos pueden ser estructurados como **procesos** en un nivel "molecular". Además de la semántica propia del sublenguaje utilizado para especificar procesos, añadiremos una semántica adicional para distinguir entre procesos de obligación (**operación**) y procesos de prohibición (**protocolo**). Una operación es un servicio de mayor nivel ofrecido por el objeto. Un caso particular de operación es cuando, además, se asume que el proceso actúa como "todo o nada" y se denomina **transacción**. Un protocolo impide la ejecución de determinadas secuencias de acciones en la vida del objeto definiendo las secuencias que están permitidas.

Cada objeto encapsula su estado y las reglas que rigen su comportamiento. Como es habitual en todo entorno OO, los objetos pueden ser vistos desde dos perspectivas distintas: estructural y de comportamiento. Desde el punto de vista es-

tructural, llamaremos **atributos** al conjunto de propiedades que describen al objeto. Los valores asociados a cada propiedad estructural del objeto caracterizan el **estado del objeto** en un instante dado. La evolución de los objetos (perspectiva del comportamiento) viene caracterizada por la noción de **cambio de estado**: la ocurrencia de una acción puede generar cambios en los valores de atributos (definidos por **evaluaciones** y **derivaciones**). La actividad de un objeto está determinada por un conjunto de reglas (propiedades de comportamiento): **precondiciones**, **restricciones de integridad**, **disparos**, **protocolos** y **operaciones**.

La **vida de un objeto** puede representarse como una secuencia de **pasos**. Cada paso está formado por un conjunto de acciones que ocurren en un instante dado de la vida del objeto.

Cada objeto tiene un identificador único (**oid**) proporcionado implícitamente por el sistema. Sin embargo, el objeto es referenciado mediante mecanismos de identificación pertenecientes al espacio del problema. Una función de identificación establece correspondencias entre los mecanismos de identificación y el *oid* del objeto.

Llamamos **tipo** a la plantilla que describe la estructura y el comportamiento común a un grupo de objetos. Una **clase** se compone de un nombre de clase, una o más funciones de identificación y un tipo. Por otra parte, una clase está formada por objetos que son **instancias**[1] de la clase. Una clase compleja es aquella definida utilizando otras clases. Las relaciones entre clases disponibles para formar clases complejas en $\mathcal{OASIS}$ son **agregación** y **especialización**.

2.1 Semántica de $\mathcal{OASIS}$

La semántica de $\mathcal{OASIS}$ es dada en términos de una estructura de Kripke (W, τ, ρ). W es el conjunto de todos los mundos[2] posibles que un objeto puede alcanzar. Sea F el conjunto de Fórmulas bien formadas (Fbf) evaluadas sobre el estado (mundo asociado) en el cual se encuentra el objeto, $\underline{A}$ el conjunto "ground" de acciones de la signatura del objeto y $2^{\underline{A}}$, el conjunto de pasos instanciados posibles. Las funciones τ y ρ se definen como:

$$\tau : F \rightarrow 2^W$$
$$\rho : 2^{\underline{A}} \rightarrow (W \rightarrow W)$$

La función τ asigna a una fórmula en la lógica de estado (Lógica de Predicados de Primer Orden) el conjunto de mundos en los cuales se satisface. La función ρ asigna a cada paso una relación binaria entre mundos, la cual es la semántica declarativa del lenguaje. Siendo $\mu \in 2^{\underline{A}}$ un paso y $w, w' \in W$, el significado buscado es: $(w, w') \in \rho(\mu)$ si y sólo si la ocurrencia de μ conduce al objeto desde el mundo w al mundo w'.

En [7] se presentan las fórmulas y especificaciones de proceso utilizadas en cada una de las secciones de una plantilla de clase $\mathcal{OASIS}$ y cómo la plantilla completa de la clase se corresponde con fórmulas en una variante de Lógica Dinámica formalizada en [9].

3 Especialización en $\mathcal{OASIS}$

Mediante especialización las propiedades definidas en las clases pueden ser refinadas. La especialización incorporada en $\mathcal{OASIS}$ está inspirada en los trabajos de Wieringa [17]. Diremos que una superclase se especializa en ciertas subclases. Una subclase hereda las propiedades definidas en la plantilla de la superclase de la que se especializa. Una subclase puede ser a su vez superclase en otra relación de especialización, de esta forma se establecen jerarquías de especialización (las cuales no deben incluir ciclos). Las propiedades de la superclase y de las subclases se especifican separadamente en sus respectivas plantillas de clase. Cuando en la subclase no existen propiedades emergentes no se define su plantilla de clase.

[1] Preferimos utilizar la palabra "instancia" en lugar de "ejemplar" por considerar que la primera, aunque es un anglisismo oficialmente no aceptado, tiene un uso más extendido.

[2] De acuerdo con lo dicho, los estados son aserciones (fórmulas), los mundos son estructuras sobre las que dichas fórmulas son interpretadas.

En $\mathcal{OASIS}$ se dispone de tres formas de especialización ortogonales entre sí, cada una de ellas con una semántica precisa:

- Particiones Estáticas.
- Particiones Dinámicas.
- Grupos de Rol.

Estas formas de especialización constituyen patrones de modelado conceptual incorporados como constructores ofrecidos por el lenguaje asociado a $\mathcal{OASIS}$. De esta forma, esta capacidad expresiva potencia el modelado centrado en el problema que se va a resolver reduciendo las dificultades asociadas a la especificación.

3.1 Particiones

Una partición establece una relación de especialización entre una superclase y las subclases formadas por subconjuntos de instancias de la superclase.

Para cada clase C, se distinguen los siguientes aspectos:

- La **intensión** de una clase, $int(C)$, es el conjunto de *todas* las propiedades que son compartidas por todas las instancias de la clase. Representa la plantilla o tipo de la clase.
- Dado un instante cualquiera t del sistema, el **conjunto existencia** (**población**) de una clase, $ext_t(C)$, es el conjunto de todas las instancias de la clase que existen en dicho instante.

Siendo C_1 y C_2 dos clases, si $ext_t(C_1) \subseteq ext_t(C_2)\ \forall t$, entonces C_1 es subclase en una partición de C_2. Cuando esto se cumple, existe una relación de inclusión inversa entre las intensiones de C_1 y de C_2, es decir, $int(C_2) \subseteq int(C_1)$.

Una partición divide **todo** el espacio de objetos de la superclase en subconjuntos **disjuntos**. Es decir, siendo $C_1, ..., C_n$ subclases de otra clase C_0 se cumple que:

$$ext_t(C_0) = \cup ext_t(C_i), \quad \forall i = 1, ..., n \tag{1}$$

$$ext_t(C_i) \cap ext_t(C_j) = \emptyset \tag{2}$$

$$\forall i \neq j, \forall i, j = 1, ..., n$$

Exigir que todas las particiones sean completas y disjuntas resuelve una serie de ambigüedades y contradicciones presentes en otras propuestas. Consideremos el ejemplo en el que `estudiante` es una subclase de `persona`. Además una persona puede ser o no, en algún momento de su existencia `estudiante`. El evento `llegar_a_ser_estudiante` debe ocurrir en la vida de la clase `persona` y no en la de `estudiante`, pero `estudiante` hereda este evento, lo que resulta en una contradicción. El hecho de que una partición sea total soluciona dicho problema puesto que en este caso se modelaría adicionalmente la subclase `no_estudiante`. Instancias de la clase `no_estudiante` tienen, en general, las mismas propiedades que `persona` y la propiedad adicional de que pueden pasar a ser `estudiante` (propiedad que `estudiante` no posee). Igualmente, objetos que sean instancias de la clase `estudiante` tienen la propiedad adicional de que pueden pasar a ser `no_estudiante` (propiedad que, de forma análoga, no posee `no_estudiante`).

En una partición cada objeto es instancia a la vez de la superclase y de una (y sólo una) de las subclases, es decir, se trata del mismo objeto (mismo *oid*). Esto tiene las siguientes consecuencias semánticas:

- Debe existir compatibilidad de comportamiento, es decir, debe cumplirse el Principio de Sustitución [16], según el cual toda instancia de la subclase debe poder usarse en el contexto de la superclase.
- Cuando el objeto es destruido en la superclase es destruido también en la subclase y viceversa.

3.2 Particiones Estáticas

En una partición estática las instancias de las subclases definidas están asociadas desde su creación a una subclase de la partición y se mantienen en ella durante toda su existencia. Es decir, siendo t_1 y t_2 dos instantes cualesquiera (con $t_1 \neq t_2$), C_i y C_j dos subclases de la partición estática (con $i \neq j$) entonces se cumple:

$$ext_{t_1}(C_i) \cap ext_{t_2}(C_j) = \emptyset \tag{3}$$

Ejemplo 1 *Dos particiones estáticas de la clase* `vehiculo`*.*

```
camion, coche, otro_vehiculo
    static specialization of vehiculo;
gasolina, diesel, otro_tipo
    static specialization of vehiculo;
```

3.3 Particiones Dinámicas

En una partición dinámica las instancias pueden migrar desde una subclase a otra. De esta forma, la intersección presentada en la fórmula 3 puede ser distinta al conjunto vacío. Se dispone de dos formas alternativas para especificar el proceso de migración: basándose en la ocurrencia de ciertas acciones o basándose en los posibles estados del objeto.

Una diferencia importante entre las particiones dinámicas y las estáticas es que en las primeras el conjunto *existencia* de alguna subclase puede cambiar sin que cambie en la superclase. En cambio, si el conjunto de existencia cambia en una subclase estática entonces cambia también en el conjunto de existencia de la superclase (por ejemplo, como consecuencia de suprimir un objeto). Así, en el ejemplo de la partición estática `vehículo` la creación de una instancia en `coche` es también la creación en `vehículo`.

Por otra parte, no se permite particionar una subclase dinámica en particiones estáticas. Esta restricción elimina complejidades innecesarias en los modelos, sin disminuir la capacidad expresiva del lenguaje.

3.3.1 Partición Dinámica Basada en Ocurrencia de Acciones

En este caso el proceso migración se define mediante una especificación de proceso que utiliza la misma sintaxis de las operaciones o protocolos de $\mathcal{OASIS}$ (esto es, un álgebra de procesos). Las acciones implicadas en la especificación de proceso pertenecen a alguna de las acciones de la subclase desde donde se migra. Las constantes agente que definen el proceso se corresponden con los nombres de las subclases de la partición. Por defecto, el **new** de las instancias es el servicio en la acción indicada al comienzo del proceso de migración.

Ejemplo 2 *Una especialización dinámica de la clase* `coche` *determinada por la ocurrencia de las acciones* `crear_coche`*,* `reparar` *y* `estropear` *puede ser:*

```
funcionando, estropeado
    dynamic specialization of coche
     migration relation is
    coche = crear_coche.funcionando;
    funcionando = estropear.estropeado;
    estropeado = reparar.funcionando;
```

Según lo dicho, la creación de una instancia de `coche` implica que el onjeto comienza su vida perteneciendo a la subclase `funcionando`. Cuando sea una instancia de `funcionando` le ocurrirán acciones de la signatura de `coche` más, posiblemente, otras de la subclase `funcionando`. Entre ellas está la acción `estropear` que pertenece a la signatura de `funcionando`. Su ocurrencia implica migrar desde la subclase `funcionando` hacia la subclase `estropeado`.

3.3.2 Partición Dinámica Basada en el Estado

En este caso el proceso de migración queda determinado por el estado de cada objeto. Cada vez que el objeto alcanza un nuevo estado esto puede implicar su migración de una subclase a otra en la partición.

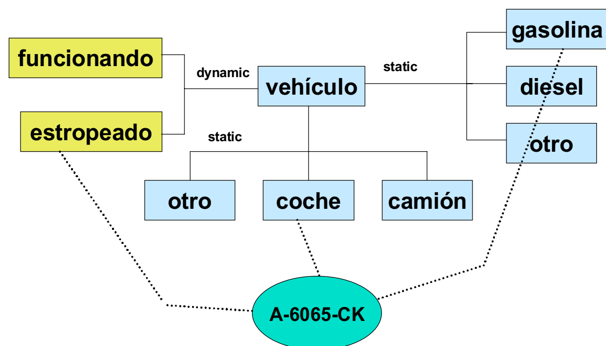

Figura 1: Particiones estáticas y dinámica sobre la clase vehículo.

Ejemplo 3 *Una partición dinámica de la clase* `cuenta` *en función del atributo* `saldo` *puede ser:*

```
no_rentable where {saldo<100000},
medio_rentable where
    {saldo>=100000 and saldo<1000000},
muy_rentable where {saldo>=1000000}
        dynamic specialization of cuenta;
```

En el ejemplo, la subclase de la partición dinámica a la cual pertenece un objeto de la clase cuenta se determina implícitamente a partir del valor del atributo `saldo`.

3.4 Especies y Herencia Múltiple

Una especie es una clase cuyo tipo es obtenido mediante una combinación de los tipos de subclases en las particiones de más bajo nivel y en una misma jerarquía de clases. Cada especie conlleva la noción de herencia múltiple de las propiedades especificadas individualmente en cada una de las clases cuyos tipos han sido combinados.

El concepto de especie aporta ventajas de tipo metodológico dado que el modelo se hace más claro y reduce las posibilidades de error.

Ejemplo 4 *La clase (especie)* `camión*diesel` *puede tener como propiedad emergente un atributo que represente la fecha del último cambio de filtro de combustible. Para ello, especificaremos la plantilla de la clase* `camión*diesel` *como para el resto de clases.*

La Figura 1 ilustra las particiones que se han definido en los ejemplos anteriores. Además se muestra cómo un objeto (cuyo valor de mecanismo de identificación es la matrícula `A-6065-CK`) es instancia de una subclase de cada una de las particiones, es decir, en este caso es instancia de la especie `estropeado*coche*gasolina`.

La herencia múltiple establece que las instancias de una clase heredan propiedades a partir de dos o más superclases. En el contexto de $\mathcal{OASIS}$ cada especie que involucre a más de una clase es ya una clase con herencia múltiple. Las propiedades emergentes que pueda tener una especie se especifican en su plantilla de clase.

Ejemplo 5 *Las especies* `funcionando*coche` *y* `coche*diesel` *son casos de herencia múltiple. El conjunto de propiedades asociadas a dichas clases es la suma de las propiedades de las clases involucradas.*

En la mayoría de las propuestas de modelado orientado a objeto se aborda la especialización múltiple de manera explícita estableciendo que una determinada clase tiene varias superclases independientes entre ellas, de esta forma se adopta una postura similar a la que toman la mayoría de los lenguajes de programación con la herencia múltiple. Esta forma de plantear la especialización es claramente menos natural que en la propuesta de $\mathcal{OASIS}$, en donde siempre existe una clase más general que ocupa la raíz de la jerarquía de especialización (con la introducción del concepto de especie se trata de un grafo dirigido acíclico). La necesidad de esta clase raíz es consecuencia de adoptar desde un principio una visión taxonómica del concepto de especialización. De esta forma si, como invitan ciertas propuestas, imaginamos una relación que vincule a una especialización con más de una superclase que no sean a su vez especializaciones de una clase más general, puede que lo que ocurra sea lo siguiente:

- Que esa clase más general exista realmente pero aún no la hemos contemplado en el

modelo, y precisamente a la hora de idear la nueva especialización nos percatamos de su existencia.

- Que la relación que queremos especificar no sea taxonómica y por tanto debemos utilizar otro mecanismo que no sea la especialización para expresarla. Esto suele ocurrir con la herencia múltiple en lenguajes de programación que en muchos casos termina utilizándose como algo parecido a una agregación.

3.5 Grupos de Rol

El uso de roles asociados a los objetos permite la representación de comportamientos diferentes de un objeto y la evolución dinámica de dicha conducta. Una clase puede tener asociada diferentes subclases de rol, cada una representando un patrón de conducta específico para un objeto de dicha subclase. El objeto sigue siendo instancia de una sola clase pero puede desempeñar distintos roles a lo largo de su existencia. De esta forma puede representarse la evolución temporal del comportamiento de un objeto.

Un rol es un objeto que tiene una relación especial con otro objeto que desempeña el rol. El objeto que desempeña el rol se denomina *player*. Una subclase de rol es aquella cuyas instancias son roles. La especialización de una clase en un conjunto de subclases de rol se denomina Grupo de Rol.

Consideraremos la función *played_by* en el modelo, tal que si R es una subclase rol, entonces existe una superclase P tal que en cada instante t se cumple que:

$$played_by : ext_t(R) \rightarrow ext_t(P)$$

Así, P representa la clase de los objetos que desempeñan el rol. Entonces, $\forall r \in ext_t(R)$, $played_by(r)$ es el objeto que desempeña el rol para r. Un grupo de rol posee las siguientes consideraciones semánticas:

- Existe exactamente un *player* asociado a cada rol.
- Pueden existir varios roles para un mismo *player*, aun cuando estos roles sean instancias de la misma subclase de rol.
- Por tratarse de objetos distintos (rol y *player*) puede no existir compatibilidad de comportamiento entre ambos. Además, la existencia del rol está supeditada a la existencia del *player*. Sin embargo, el *player* puede seguir existiendo cuando un rol asociado es destruido.
- Cada grupo de rol representa un conjunto de subclases de rol mutuamente exclusivas, es decir, de forma simultánea un *player* puede desempeñar roles de como máximo una clase de rol dentro de cada grupo de rol.
- A diferencia de las particiones puede haber instancias de un *player* que no desempeñen ningún rol en un grupo de rol. Esto permite que un grupo de rol pueda tener una sola subclase de rol.
- Se pueden definir particiones estáticas y dinámicas o nuevos grupos de rol a partir de una subclase de rol.

Además, es importante destacar que las subclases heredan tanto las propiedades de la superclase como los roles que se pueden desempeñar. Para el ejemplo, una instancia de la clase `estudiante` puede desempeñar el rol de `empleado`.

Ejemplo 6 *Un ejemplo de un grupo de rol definidos sobre la clase* `persona`*.*

```
estudiante towards(0,1)
            ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

Se ha querido representar con las cardinalidades el que una instancia de la clase `persona` puede desempeñar, simultáneamente, como máximo `10` *roles* de `empleado` o, alternativamente, como máximo un rol de `estudiante`. También es posible que una instancia de `persona` no desempeñe ningún rol.

El evento de creación es enviado a la clase `persona`. Posteriormente, si ocurre el evento `ser_matriculado` (que representa el **new** de `estudiante`), entonces la `persona` pasa a desempeñar el rol de `estudiante`. Si destruimos el objeto de `persona`, automáticamente se destruye el objeto `estudiante`, lo contrario no se cumple.

Ejemplo 7 *Consideremos la partición dinámica de* `persona` *junto a los roles definidos sobre la misma clase en el ejemplo anterior:*

```
niño where {edad<14},
adolescente where
       {14<=edad and edad<18},
adulto where {18<=edad}
    dynamic specialization of persona;

estudiante towards(0,1)
            ser_matriculado,
empleado towards(0,10) ser_contratado
    role of persona;
```

En este caso, la existencia del rol no supone aumentar el número de especies dado que es un nuevo objeto el que se crea cuando se comienza a desempeñar el rol. En cambio, a futuras particiones (estáticas o dinámicas) de la clase de rol sí podrían dar lugar a nuevas especies.

4 Ejemplo

En este apartado estudiaremos el ejemplo de la Figura 2, mostrado en notación UML. De acuerdo al análisis de las interpretaciones posibles para el ejemplo podremos ilustrar los aportes del enfoque propuesto e integrado en *OASIS*.

La Figura 2 muestra una especialización de la clase `Cuenta` en las subclases `Cuenta de Crédito` y `Cuenta de Ahorro`. En lo que respecta a reutilización y extensibilidad de la superclase `Cuenta`, el objetivo se ha conseguido, pero ¿cuál es la semántica asociada?. La única caracterización provista es la determinada por la restricción `{disjunta, completa}`. Para responder a la pregunta anterior recurriremos a la caracterización semántica de particiones y grupos de rol en *OASIS*.

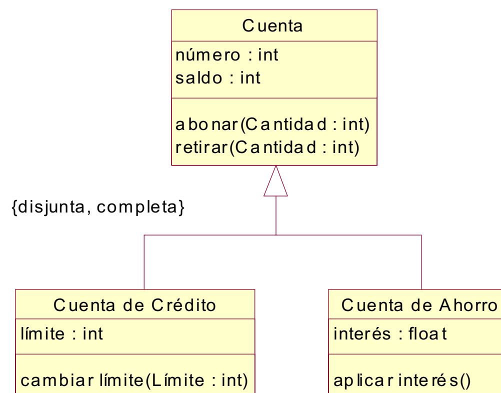

Figura 2: Generalización usando UML

Para comenzar, debemos recordar que en *OASIS* tanto las particiones como los grupos de rol son disjuntos. Por otra parte todas las particiones son completas (aunque gráficamente podría evitarse el poner la subclase "otros" y poner en su lugar la decoración `{incompleta}`). Sólo en el caso de grupos de rol la jerarquía puede ser incompleta.

Caso I Consideremos las condiciones siguientes:

- En una posible redefinición de las propiedades de la clase `Cuenta` en las subclases siempre se respeta la compatibilidad de comportamiento entre los objetos como instancias de la subclase y de la superclase
- La destrucción de una instancia de `Cuenta de Crédito` o de `Cuenta de Ahorro` implica la destrucción de la instancia en la clase `Cuenta`
- Una instancia de `cuenta` puede ser una solo instancia en una de las subclases

Si se cumplen TODAS, entonces se trata de una partición. A continuación debe decidirse si se trata de una partición dinámica o estática. Si una instancia `Cuenta de Crédito` o `Cuenta`

de Ahorro puede migrar a la otra subclase entonces se trata de una Partición Dinámica, sino, es una Partición Estática.

En caso de ser una Partición Dinámica queda por especificar el proceso de migración. Si dicho proceso está basado en ocurrencia de acciones el siguiente podría ser un ejemplo de proceso de migración:

```
Cuenta = alta.Cuenta  Ahorro
Cuenta de Ahorro =
    convertir_a_crédito.Cuenta de Crédito
Cuenta de Crédito =
    convertir_a_ahorro.Cuenta de Ahorro
```

En esta situación los eventos `convertir a crédito` y `convertir a ahorro` serían definidos en las subclases `Cuenta de Ahorro` y `Cuenta de Crédito`, respectivamente. Si el proceso migratorio estuviese basado en estado el siguiente podría ser un ejemplo de su especificación en $\mathcal{OASIS}$:

```
Cuenta de Ahorro where {saldo>=0},
Cuenta de Credito where {saldo<0}
    dynamic especialization of cuenta;
```

Caso II No se cumple alguna de las condiciones del Caso I, es decir, alguna de las siguientes afirmaciones es verdadera:

- Una redefinición de las propiedades de la clase `Cuenta` en la subclases puede hacer que los comportamientos sus instancias no sean compatibles en la subclases con respecto de la superclase
- La destrucción de una instancia en las subclases no implica necesariamente la destrucción de la instancia en la clase `Cuenta`
- Más de una instancia de una subclase puede estar asociada a la misma instancia de `Cuenta`

Por lo tanto, se trata de un Grupo de Rol. Queda por determinar el evento que implica la creación del rol y la multiplicidad mínima y máxima permitidas. Por ejemplo, una especificación $\mathcal{OASIS}$ en este caso podría ser:

```
Cuenta de Crédito towards(0,1)
    convertir_a_crédito,
Cuenta de Ahorro towards(0,1)
    convertir_a_ahorro
    role of Cuenta;
```

Los eventos `convertir a crédito` y `convertir a ahorro` serían operaciones de la clase `Cuenta`. Según lo especificado, si el interés en el ejemplo fuera el poder prescindir de la compatibilidad de comportamiento, en las definiciones de las subclases estaría permitido redefinir propiedades de la clase `Cuenta`, manteniendo la compatilidad sólo a nivel de signatura.

Como se observa al análizar ambos casos los constructores de $\mathcal{OASIS}$ determinan con precisión la semántica de la especialización. Obviamente esto no descarta el uso de notaciones gráficas. La notación de UML podría ser utilizada añadiendo ciertos estereotipos (por ejemplo: <<`estática`>>, <<`dinámica`>> o <<`grupo de rol`>>) y otras decoraciones como condición de especialización (como una restricción cuando se trata de una partición dinámica basada en el estado), evento de especialización o multiplicidad, estos últimos en el caso de roles. Lo destacable es que en nuestra propuesta se guía al analista en el proceso de modelado, exigiéndole que seleccione el constructor adecuado de acuerdo con la semántica perseguida.

5 Trabajos Relacionados

En el contexto de los enfoque semiformales más utilizados en el modelado orientado a objetos podemos tomar como referencia la notación UML. La llamada generalización en UML ofrece una escasa caracterización semántica y la herencia asociada queda libre a distintas interpretaciones tal cómo se demostró con el ejemplo de la sección anterior. En algunos trabajos asociados a UML se ha reconocido la necesidad de representar aspectos tales como: particiones [8], diferenciar entre subclasificación (particiones) estáticas y dinámicas [4], y el concepto de roles [1]. Pero en estos casos sólo se añaden decoraciones a la notación sin presentar un soporte formal que determine

con precisión la semántica asociada.

Respecto de las propuestas formales con una motivación similar a la de $\mathcal{OASIS}$ son OBLOG [10], TROLL [6], TESORO [15], ALBERT [2] y LCM [3]. La principal deficiencia respecto de la especialización (herencia) es que en la mayoría de los enfoques contrastados el modelado de las relaciones de especialización (cuando se puede realizar explícitamente) debe hacerse al mismo nivel de abstracción ofrecido en un lenguaje de programación orientado a objeto. En OBLOG, TROLL y TESORO el mecanismo de especialización no se orienta tan directamente hacia situaciones de modelado conceptual como en $\mathcal{OASIS}$. En ALBERT y LCM la especialización es soportada por relaciones entre tipos pero no se proporcionan operadores específicos para el modelado. En ninguno de los enfoques comparables a $\mathcal{OASIS}$ existe posibilidad de definir especializaciones simultáneas de un objeto, ni las restricciones de multiplicidad asociadas. Tampoco se provee un soporte formal para definir varias especializaciones a partir de una misma clase. Sólo en $\mathcal{OASIS}$ es posible especificar un proceso de migración de objetos entre subclases. Por último, el tratamiento de la herencia múltiple no es abordado tan detalladamente como se hace en $\mathcal{OASIS}$ y TESORO.

Además, es importante destacar que a diferencia de otros enfoques formales, en $\mathcal{OASIS}$ desde sus primeras versiones se ha puesto especial interés en tres aspectos esenciales que facilitan su integración en entornos industriales de desarrollo de software[3]:

- Establecimiento de un método de desarrollo basado en $\mathcal{OASIS}$, llamado OO-METHOD
- Generación automática de código a partir de especificaciones $\mathcal{OASIS}$, orientada tanto a la validación y como a la construcción del producto final
- Construcción de herramientas que soporten el proceso de modelado conceptual. Se ha construido un entorno CASE basado en la versión precedente de $\mathcal{OASIS}$ y adaptado a OO-METHOD

[3]Para obtener un detalle de los trabajos mencionados consultar la dirección: `www.dsic.upv.es/users/oom`

El segundo punto ha sido determinante en cuanto a la forma de tratar la especialización en $\mathcal{OASIS}$ puesto que era necesario que ésta fuese soportada en forma precisa en lenguajes de programación y mediante procesos de generación automática de código.

6 Conclusiones y Trabajo Futuro

En este artículo se ha presentado una visión resumida de la especialización y su integración dentro del marco de modelado conceptual formal que proporciona $\mathcal{OASIS}$.

Los constructores conceptuales presentados en este trabajo aparecen también como una necesidad en diseño. Concretamente en el área de los patrones de diseño existen patrones para implementar la clasificación dinámica (patrón *state* en [5]) y los roles (patrón *role object* en [1]). Esto evidencia que los diseñadores necesitan de construcciones de mayor nivel de abstracción, no soportadas directamente por los lenguajes de programación actuales. Así, las ideas presentadas en este artículo, situadas en el ámbito del modelado conceptual tienen su contrapartida en el diseño. En este sentido estamos trabajando en el refinamiento de un proceso de traducción automática que partiendo de los constructores conceptuales y mediante determinadas correspondencias con patrones de diseño permiten obtener la implementación. Asimismo, se está trabajando en el tratamiento de la especialización y las características de la herencia en el marco de la Lógica Dinámica [12].

Como posibles líneas de trabajo futuro están: la definición de un procedimiento para verificación automática de compatibilidad de comportamientos dentro del marco $\mathcal{OASIS}$ y el aprovechamiento de la semántica proporcionada por los mecanismos aquí presentados para dar soporte formal a propuestas semiformales como UML.

Referencias

[1] Baümer D, Riehle D, Siberski W. and Wulf M. *Role Object*. In Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP '97). Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997.

[2] Dubois E., Du Bois P. and Petit M. *O-O Requirements Analysis: an agent perspective*. In Proc. of the 7th.European Conference on Object Oriented Programming (ECOOP'93), pages 458-481, 1993.

[3] Feenstra R.B. and Wieringa R.J. *LCM 3.0: a languaje for describing conceptual models*. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[4] Fowler M and Scott Kendall. *UML Destilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[5] Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[6] Jungclaus R., Saake G., Hartmann T. and Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*. ACM Transactions on Information Systems, Volume 14, Number 2, pages 175-211, 1995.

[7] Letelier P., Ramos I., Sánchez P. y Pastor Ó. *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*. Servicio de Publicaciones de la Universidad Politécnica de Valencia, ISBN 84-7721-663-0, Universidad Politécnica de Valencia, 1998. `http://www.dsic.upv.es/users/oom-/books.html`.

[8] Martin J. and Odell J. *Object-Oriented Methods: A Foundation*. Prentice Hall, 1998.

[9] Meyer J.-J.Ch. *A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic*. In Notre Dame Journal of Formal Logic, Volume 29, pages 109-136, 1988.

[10] OBLOG Software S.A. *The OBLOG Software Development Approach (White Paper)*. Abril 1999. `http://www.oblog.pt/Download-/Documentation.exe`.

[11] Rumbaugh J., Jacobson I. and Booch G. *The Unified Modeling Language: Reference Manual*. Addison-Wesley. 1999.

[12] Sánchez P., Letelier P., Ramos I. *Animating Formal Specifications with Inheritance in a DL-Framework*. Por aparecer en Requirements Engineering Journal, Springer-Verlag.

[13] Taivalsaari A. *On the Notion of Inheritance*. ACM Computing Surveys, 28(3), pages 438-478, September 1996.

[14] Troyano J.A. *Herencia y Clasificación en un Lenguaje de Especificación Orientado a Objetos*. Tesis Doctoral, Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla, Junio 1998.

[15] Torres J. *Especificaciones Orientadas a Objetos basadas en Restricciones. Prototipado en un Lenguaje Orientado a Procesos*. Tesis Doctoral, Departamento de Lenguajes y Sistemas Informáticos de la Univerisdad de Sevilla, Diciembre 1997.

[16] Wegner P. and Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. LNCS 322, ECOOP pages 55-77, 1988.

[17] Wieringa R., Jonge W. and Spruit P. *Using Dynamic Classes and Role Classes to Mo-*

del Object Migration. Theory and Practice of Object Systems, 1(1), p.61-83, 1995.